C/C++ Memory Model and Interactions

Project 1

Alex Harper

Contents

List of Figures

1 Introduction

The assignment of this project was to learn how C works with memory on computers. I have been coding in C++ for a long time and, mostly out of interest, done a lot of reading into how other projects and languages do things. C keept things simple, and C++ inherited that simplicity but tried to change the paradim. I go through the basic ideas and usages of C style memory managment first, and then talk about how C++ does similar thigns but compatable with objects.

2 Simple C

The simple language that many harolded as the "portable assembler" tried to be simple yet generic about how it operated. The concept of memory in the language is simply a single dimension of bytes. Think of a string, you have one end that you can call the begining, and the other end you call the end. Imagine that every inch of the string is a single byte, and how many inches you have is how many bytes of memory you have to work with. The first byte you can read or write is 0 inches away from the beginning of the string, so we can call it byte 0. If we want to write a value to the fith byte, then the start of that section of string is 4 inches away from the beginning.

Address	0	1	2	3	4	5	...
Bits	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7	

Figure 1: Imaginary String With Sections

2.1 Simple Pointers

Pointers are the way we tell the language what address we are using. Normal variables don't make us know anything with where they are located, they simple "hold a value". Pointers dont "hold a value" but instead "hold an address". You can see some code in fig2 that does the basic operations of pointers.

The first section is the normal way of doing things; declaring a variable, assiging it a value, printing it out. It makes perfect sense that it prints out 10 given a variable holding the number 10. The second part is using a pointer to read the value of a; declare a pointer with the name p, give p the address of a, read the location that p points to. p is given the address of a (`p=&a;`), not the value of a. When we go to print the value, we must *dereference* the address. Using ***p** is the same as using **a**

Code	Output

`int a;`
`a = 10;`
`printf("%d\n",a);`
 	
`int *p;`
`p = &a;`
`printf("%d\n",*p);`
 	
`*p = 25;`
`printf("%d\n",a);`
	10
	10
	25

Figure 2: Simple Pointer Operation

directly. Since both *dereferencing* and using **a** directly is the same, we can do what is in the third section. Using the pointer, it gets dereferenced, and then a new value assigned to it. Since we wrote to the same spot in memory that **a** lives at, **a** now has the new value.

2.2 Walking The Memory With Pointers

Arrays are simple groups of variables. You say how many things of a certain type you want, and you get an array that is big enough to hold them all. Pointers though only reference to a single spot in memory, but, pointers are also just numbers! Remember that an address is really just a number saying how far from the beginning of memory. So since it is a number, can't we do math with it? Yes, and with simple addition and subtraction, we can move around arrays!

The example code in fig3 starts with our basic array of integers. We give the 4 indicies the numbers 2,5,8,11 to start with. The first print statment shows 3 different pointer uses; simple dereference, adding to the value, adding to the position. The first number that comes out is 2, because all we are doing is reading the value at the first index. The second number is 3 because we read the value and *then* add 1 to it. The third number is 5 because we add one to the *position* and then read the value. That last way is the same as saying `b[0+1]`. If we instead want the 3 index, either we can do `b[0+2]` or equivilently `*(p+2)`.

The last section just drives home how adding to the pointer is basicly the same as adding to the index in the square brackets doing it the normal way. It adds x to the position, and sets the value to something new. So what does this tell you about **b**? Looking at the line `p = b`, you should notice that we didn't use the **&** this

time. That is because b is already a pointer! The language just hides it from you, doing the conversion of the square brackets. So typing `b[3]=11;` is transparently translated to `*(b+3)=11;`.

Code	Output

`int b[4];`
`b[0]=2; b[1]=5; b[2]=8; b[3]=11;`
 	
`int *p = b;`
`printf("%d %d %d\n",*p,(*p)+1,*(p+1));`
 	
`for(int x=0; x<4; x++)`
`    *(p+x) = 4*x;`
`printf("%d %d %d %d\n",b[0],b[1],b[2],b[3]);`
	`2 3 5`
	`0 4 8 12`

Figure 3: Walking An Array Basics

If you play around with pointers for a short time, you will probably try something like in fig4. The first section is making the array and having **p** point to it like before. The difference is that instead of adding to **p** when printing, it is permantly changed before then. When it gets printed now, it prints the third index even though we dont add to it during the print. This is because **p** now points farther into the array than just the beginning.

But the more interesting thing is for the second section of fig4. It does not matter what the type of the thing we are pointing at is. We have an array of size 50, and we set the contents to the value of a string `"Some Length"`. Since we are lazy and dont want to count ourselves the number of characters in our string, we instead march a pointer down the length of it with a counter. Strings in C are expected to *always* end with a null terminator, aka the number 0; So the loop keeps going until the pointer points to a value of zero, with every itteration adding 1 to the counter and adding 1 to the pointer. When we print out the counter, we get 11, which is how many characters are in the string. Well probably, as I said, I am lazy and just expect the computer to get it right.

2.3 Scoping Still Applies

So now that we have learned the basics of moving pointers around, there is an important thing to talk about. Pointers keep their address that they point to even when the variable they point to does not exist any more. Look at fig5 for some really simple code to show it going wrong. The second part tries to get an address from the function. The function, wrongly, returns the address of a local variable. When the

Code	Output

`//pointers can themselves be changed`
`int b[4];`
`b[0]=2; b[1]=4; b[2]=6; b[3]=8;`
`int *p = b;`
`p += 2;`
`printf("%d\n",*p);`
 	
`//can be used to walk arrays of characters`
`char s[50]="Some Length";`
`char *sp = s;`
 	
`int counter=0;`
`while(*sp != 0){`
`    counter+=1;`
`    sp += 1;`
`}`
`printf("%d\n",counter);`
	6
	11

Figure 4: Walking An Array Advanced

first function returns the address, it also deletes the variable. You should be familure with scoping rules in C style languages; you leave the enclosed space of curly braces, you leave their scope. Anything that is localy created in any scope will get deleted when leaving said scope.

Code	Output

`int* bad_function(){`
`    int x=25;`
`    return &x;`
`}`
 	
`int *p=__scope_of_variables();`
`printf("%d\n",*p);`
	`Segmentation fault`

Figure 5: Accessing Out Of Scope Variables

The Compiler knows about these problems and tries to even be helpful. Below is the message I get when I compile the above code.

```
examples.cpp:83:6: warning: address of local variable x returned [-Wreturn-local-addr]
  int x=25;
      ^
```

So now you might be thinking that you can tackle any task dealing with pointers, but there is a dark side to them. The issue is insideous and full of terror, something that makes people lay awake at night once they deal with it the first time, **Coincidental Behavior**. Look at fig6 and notice how the pointer points to the just deleted variable (curly braces make a nested scope). Then notice that the output has the right number! While, yes, this code does do the right thing, it is also wrong.

The pointer accessing the address of the freshly deleted variable **x** is part of what is called *Undefined Behavior*. Actually, a pointer pointing to anywhere in memory that is not currently valid or you don't own is undefined behavior. Just by chance, the code works this time, but on a different system, in a different context, it might not work. Using some flavor of undefined behavior and not noticing is not always the end of the world, but it can often lead to odd results during certain conditions. That is why things working by coincidence is scary; it works today, but maybe not tomorrow, and when it fails, it is often subtle as to the cause.

(Just to note, *Undefined Behavior* is not just about pointers, but about anything the standard of the language does not specify)

Code	Output

`int *p;`
`{ // some other scope`
`  int x = 100;`
`  p = &x;`
`}`
`printf("%d\n",*p);`
	100

Figure 6: Accessing Out Of Scope Variables But This Time It Works

2.4 Getting Your Own Space

After scaring you with scoping issues, lets look at how we can get around such things. This is where system calls such as `malloc()` com in. These functions do some magic behind the scenes and ask the kernel "Hey, I need X number of bytes to use". The kernel (usually) will hand back the answer of "Here is the address of your space to use". The address points to the first byte of a space that is X bytes long. Since it is just a length of bytes, we can pretend it is an array! In fig7 is some of the most basic code you can do.

So what is the point of getting such a space? The space is entierly yours! Pass it around, write randomly in it, move things around, whatever you want! When does it disapear? When you want! Scoping means nothing to this piece of memory!

Code	Output

`int *p = (int*)malloc(4*sizeof(int));`
`for(int x=0; x<4; x++)`
`    p[x] = x*4;`
`printf("%d %d %d %d\n",p[0],p[1],p[2],p[3]);`
`free(p);`
	`0 4 8 12`

Figure 7: Simple malloc() Array

So lets fix the previous example where it had a segfault using fig8. It simply allocates enough space for a single integer, assigns a value, passes back the address, and then prints.

Code	Output

`int* malloc_single(){`
`    int *p = (int*)malloc(sizeof(int));`
`    *p = 25;`
`    return p;`
`}`
`int *p = malloc_single();`
`printf("%d\n",*p);`
`free(p);`
	`25`

Figure 8: Working Around Scope With malloc()

When using **alloc()* calls, there are a few things to know about. Just because you left the scope of where you allocated the memory does not mean that memory goes away. This lets you do things like allocate an array, fill it, and then return the results back. To make the allocated space actually go away, you have to use the *free()* function. You take the pointer that malloc handed you, and you give it to *free()*. Then the memory is gone! This is a place where you have to be careful though, since you need to make sure you never access the same place again with a stale pointer somewhere. Also, there is the opposite problem of having memory allocated but no pointer pointing at it. This memory can essentailly be considered lost, and we call that a *memory leak*. If your program has a memory leak, then as it runs, it might allocate all the RAM on the system even though it is not usign it.

2.5 The Right Call For The Job

There are 3 functions to allocate memory, each do a different thing for a different job; *malloc()*, *calloc()*, *realloc()*. While you can do everything with the simple *malloc()*, the other 2 provide a couple extra features for specific jobs.

2.5.1 malloc()

```
void *malloc(size_t size);
```

This is the simplest of the 3, where you ask for a number of bytes and you are handed back a pointer. As I have already done examples above with *malloc()*, I will forgo using the sapce here. An import note is that the memory is uninitiallised, meaning you dont know what values will already be stored there.

2.5.2 calloc()

```
void *calloc(size_t nmemb, size_t size);
```

This is kindof the same as *malloc()* in that it gives you some space to work around in, but is different in that it is more array oriented. You pass it two arguments; number of elements, and size of each element. What it hands back is enough room to hold everything you just asked for, just like an array. Example in fig9.

Code	Output

`int *p = (int*)calloc(4,sizeof(int));`
`for(int x=0; x<4; x++)`
`    p[x] = x*4;`
`printf("%d %d %d %d\n",p[0],p[1],p[2],p[3]);`
`free(p);`
	`0 4 8 12`

Figure 9: Example Usage Of calloc()

You can do the same thing with *malloc()* by doing `malloc(nmemb * size)`. The main difference is that *calloc()* does not return uninitialized memory and instead sets all the bytes to 0 first. This can be important as it makes sure your memory is in a known state when you use it if you don't initialize it yourself.

2.5.3 realloc()

```
void *realloc(void *ptr, size_t size);
```

This is a more special function of the 3. The use for this is not to (neccisarily) allocate memory, but to chage the size of the allocation you already got. Assume you

already have allocated memory for an array, but then are asked to add more things to the array. Using *realloc()*, you simply give it the pointer to the current memory, ask it to make it a given size, and use the pointer it gives back to you. It handles copying all the data over to the new memory. Example in fig10.

Code	Output

`int *p = (int*)calloc(2,sizeof(int));`
`p[0]=1; p[1]=1;`
 	
`p = (int*)realloc(p,20*sizeof(int));`
`for(int x=2; x<20; x++)`
`    p[x] = p[x-1] + p[x-2];`
`printf("%d\n",p[19]);`
`free(p);`
	6765

Figure 10: Example Usage Of realloc()

What happens is realloc tries to make the current block larger and return the same pointer back. If you are asking it to shrink the block to be smaller than it currently is, it always will return the same pointer. If you ask it to make the block larger, then it might need to move to a different part of memory. When it returns the pointer to you and it is different, you just use the new pointer. The data will have been copied over, and the old pointer will already have been *free()*ed.

A few notes about realloc:

- If it grows a block, the new section will be uninitialized
- If it moves a block, it copies the whole od block over even if you were not using the whole old block
- If you ask for a size of zero, it will actually free the pointer you give it
- If you give it a pointer *NULL* and some size, it acts the same as *malloc()*

2.5.4 free()

As you should already have picked up on, this does the exact opposite of the other commands. After you are done using a block of memory, you used the *free()* function to release it back to the OS. You must pass the pointer that the other function gave to you, not some pointer in the middle of the block.

You can pass in *NULL* to it, and it does nothing in that case. This is great for automatic cleanup code; you free the pointer, set the pointer to *NULL*, don't worry if you call free on that pointer again.

2.6 Honorable Mentions

Some things that I consider dealing with memory but are not often used, have specific use cases, more advanced than I want to delve into right now, or too dangerous to use.

- *volitile* - This tells the compile that it can't optimize any operations with a variable. Usually this is a bad idea and the reasons to use it are typically for things like kernel and driver code.
- *reallocarray()* - Similar relationship that *calloc()* has with *malloc()*, except with *realloc()*
- *alloca()* - Allocates memory on the stack instead of the heap. Basicly means that when you return from the function that it was made in, it becomes invalid. Basicly the same as a local array variable.
- *mmap()* and *munmap()* - Maps files and devices into memory so that writes and reads can be done with pointers.
- *mallopt()* - Changes the behavior of *malloc()*.
- *getpagesize()* - Returns the size of a single page of memory is. Returns 4096 on my system. Seems it is not even implemented on all systems these days.
- *sysconf()* - Get information about the system at run time. Depending on what paramter you pass in determines what information it returns.

2.7 Showing Off

Just to show off some, I made a dynamic array thing in C. While the API is a little rough, it does make sure that I am doing things the right way.

```c
struct DynamicArray{
    void *data;
    int capacity;
    int length; // here for my own convenience
};
```

Figure 11: Data Structure For The Dynamic Array

```c
struct DynamicArray* make_array(int size){
    if(size<=0)
        size = 1;
    struct DynamicArray *info =
        (struct DynamicArray*)malloc(sizeof(struct DynamicArray));
    info->data = malloc(size);
    info->capacity = size;
    info->length = 0;
}
void delete_array(struct DynamicArray* info){
    free(info->data);
    free(info);
}
```

Figure 12: Functions To Create And Delete The Array

```c
void resize_array(struct DynamicArray* info,int size){
    if(size==0) size = 1;
    if(size<0){
        int delta = info->capacity;
        if(delta > 128) delta = 128;
        info->data = realloc(info->data,delta+info->capacity);
        info->capacity += delta;
    }else{
        info->data = realloc(info->data,size);
        info->capacity = size;
    }
}
```

Figure 13: Function That Resizes The Array

Code

```
//Make array and fill with 50s
struct DynamicArray *array = make_array(10*sizeof(int));
for(int x=0; x<10; x++)
    ((int*)array->data)[x] = 50;
array->length = 10; // just to help me bookkeep

//lets just check that things are working as we expect
//print all 10 indicies
for(int x=0; x<array->length; x++){
    printf(" %3d ",((int*)array->data)[x]);
}
printf("\n");
```

Output

```
 50  50  50  50  50  50  50  50  50  50
```

Figure 14: First Dynamic Array Test: Simple Use

Code

```
//now lets make the array larger
resize_array(array,30*sizeof(int));
for(int x=0; x<10; x++)
    ((int*)array->data)[x+10] = 100;
for(int x=0; x<10; x++)
    ((int*)array->data)[x+20] = 150;
array->length = 30; // just to help me bookkeep

print_array(array);
```

Output

```
 50  50  50  50  50  50  50  50  50  50
100 100 100 100 100 100 100 100 100 100
150 150 150 150 150 150 150 150 150 150
```

Figure 15: Second Dynamic Array Test: Resize Larger

Code

```
//now lets make the array smaller
resize_array(array,20*sizeof(int));
array->length = 20; // just to help me bookkeep
print_array(array);
```

Output

```
 50  50  50  50  50  50  50  50  50  50
100 100 100 100 100 100 100 100 100 100
```

Figure 16: Second Dynamic Array Test: Resize Smaller

3 C++ Doing Objects

Now that we have gone through the way C does things, I think it is prudent to look at C++. The ideas are mostly the same, with memory being 1-dimensional and scope being something to look out for. The thing that makes C++ really different from C is having *Objects*. While a bit obvious, there are 2 main things about working with objects that set it apart from regular C; creating/destroying and how to use/encapsulate the memory.

3.1 Allocating Objects

For the most part, there are only 2 new keywords to remember; *new* and *delete*. When doing C++, often you will want a more permant object than just what is provided in your scope. Unfortuantly, *malloc()* and family don't play well with objects.

When you allocate a block for, lets say, integers, all you get back is a region of memory to use. That is fine for such simple things, but objects have a bit more to them that runs *implicitly*. Namely constructors and destructors are what gets missed doing it the C way, hence getting 2 new keywords. Looking at fig17 you see a simple object that prints when it gets constructed and destructed. Well, it is supposed to be, but the *malloc()* seems to not do such things!

Code

```
1 class Simple{
2 public:
3     Simple(){printf("Constructed\n");}
4     ~Simple(){printf("Destructed\n");}
5 };
6
7 Simple *p = (Simple*)malloc(sizeof(Simple));
8 free(p);
```

Output

Figure 17: Wrongly Allocating Objects

Seeing that it doesn't work the old way, lets try with the new keywords in fig18. The output clearly shows the text being printed, so it works as expected. In reality, it is mostly doing the same thing as *malloc()* except it also adds in calls to the constructor and destructor.

As you probably would expect from something that is basicly a fancy *mal-*

Code

```
class Simple{
public:
    Simple(){printf("Constructed\n");}
    ~Simple(){printf("Destructed\n");}
};

Simple *p = new Simple;
delete p;
```

Output

```
Constructed
Destructed
```

Figure 18: Rightly Allocating Objects

loc(), the allocated object outlives the scope it was created in. In fig19 we do the normal test of returning the address of something from another function and see if it segfaults. As is evident in the output, it works just as expected and all is good.

Code

```
class HelloWorld{
public:
    void print(){printf("Hello World\n");}
};

HelloWorld* some_other_scope(){
    return new HelloWorld;
}
HelloWorld *ppp = some_other_scope();
ppp->print();
delete ppp;
```

Output

```
Hello World
```

Figure 19: new Objects Also Outlive Their Scope

We are able to allocate a single object, but for true feature parity we must be able to allocate arrays too. A little syntactical sugar comes in the form of *new[]* and *delete[]*. The *new[]* tries to emulate the normal way of declaring an array and the *delete[]* basicly works the same as before. A simple example of classes that count themselves is shown in fig20. Only one thing to remember, if you use *new[]*, you should use *delete[]* even though *delete* doesn't throw any errors.

Code

```
class Counter{
    static int counter;
    int our_value;
public:
    Counter(){
        our_value = counter;
        counter++;
    }
    void print(){printf(" %d ",our_value);}
};
int Counter::counter = 0;

Counter *array = new Counter[5];
for(int x=0; x<5; x++)
    array[x].print();
printf("\n");
delete[] array;
```

Output

```
 0  1  2  3  4
```

Figure 20: Allocating Arrays Of Objects

3.2 malloc() Still Is Usefull In C++

Even with the new keywords that make objects behave properly, that doesn't mean *malloc()* is superseded. The best example I can think of is (once again) a dynamicly sized array. Having such a thing is useful in programming, every language has something, and in C++ it is called *std::vector*. Encapsulating the sometimes tricky bits of memory managment is what objects are great at. The following snippets show a basic implementation of such a thing, and that *malloc()* and *realloc()* is a key part of it.

The code in figures 21 22 23 is the implementation of such an array thing. It is an object that will grow itself if you try to push too many items onto the back of it. It starts with enough room to hold 1 item internally, and so nearly immediately will need to grow the block size to accommodate more. When it grows, it initially doubles its capacity, but if it would be an increase of more than 128, it limits the increase to 128 more items. To make it smaller, all it needs to do is subtract frrom the internal counter of how long it is.

In fig24 you see a simple test of the object holding integers. Running through the 3 main ways of changing the data, it seems to hold up just fine. The *realloc()* is

```
template <typename T>
class vector{
    T* data;
    int length,capacity;
public:
    vector(){
        data = (T*)malloc(sizeof(T));
        length = 0;
        capacity = 1;
    };
    ~vector(){delete[] data;}
    void push_back(T& item);
    void pop_back();
    T& operator[](int index);
};
```

Figure 21: Class Definition For Dynamic Array Object

```
template<typename T>
void vector<T>::push_back(T& item){
    if(length >= capacity){
        //need more room, time to realloc()
        int delta = capacity;
        delta %= 128; //only get bigger by steps of 128 at most
        data = (T*)realloc(data,(delta+capacity) * sizeof(T));
        capacity += delta;
    }
    data[length] = item;
    length++;
}

template<typename T>
void vector<T>::pop_back(){
    length--;
}
```

Figure 22: Dynamic Add And Remove Methods

```
template<typename T>
void vector<T>::pop_back(){
    length--;
}
```

Figure 23: Array Access Method

handeled transparently, but is still a vital part of how the object works.

Code

```
vector<int> test;
//add items
for(int x=0; x<10; x++)
    test.push_back(x);
//read items
for(int x=0; x<test.size(); x++)
    printf(" %d ",test[x]);
printf("\n");

//remove items
for(int x=0; x<5; x++)
    test.pop_back();
print_items(test);

//change items
for(int x=0; x<5; x++)
    test[x] = (x+1) * 5;
print_items(test);
```

Output

```
 0  1  2  3  4  5  6  7  8  9
 0  1  2  3  4
 5  10  15  20  25
```

Figure 24: Testing The Vector With Simple Integers

In fig25 is the same test being done with the previously made Counter object. It mostly does the same things for the test, but it shows that it handles constructing objects correctly.

Code

```
class Counter{
    static int counter;
public:
    int our_value;
    Counter(){
        our_value = counter;
        counter++;
    }
    void print(){printf(" C%d ",our_value);}
};
int Counter::counter = 0;

vector<Counter> test;
//add items
for(int x=0; x<10; x++)
    test.push_back(Counter());
print_items(test);

//remove items
for(int x=0; x<5; x++)
    test.pop_back();
print_items(test);

//change items
for(int x=0; x<5; x++)
    test[x].our_value = (x+1) * 5;
print_items(test);
```

Output

```
C0  C1  C2  C3  C4  C5  C6  C7  C8  C9
C0  C1  C2  C3  C4
C5  C10  C15  C20  C25
```

Figure 25: Testing The Vector With Objects

A Mini-Examples Code Used For Double Checking

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

//1-11 = 16:00 to 17:30 - move files in git and setup readme files
//1-12 = 18:00 to 18:30 - start on example.cpp
//1-13 = 19:00 to  2:30 - make outline of example.cpp, get basic things ready, simp
//1-14 = 00:00 to  4:00 - work on examples and report
//1-16 = 12:00 to  0:30 - finish C part of the paper (-30 mins for lunch)(-30mins t

/* topics

simple pointers in a 1d memory
pointers to walk an array
    going out of bounds of the array
pointers are invalid outside of their scope
    show still get right value when read from invalid location when ar leaves the s
    explain why this is bad
malloc() - a "replacment" for global variables
    gives back memory that your program holds until the program ends
    free() - tell the OS that we dont need this memory any more
calloc()
realloc()
    what it does
    example making a dynamicly sizing array (like std::vector) with a struct and fu
volatile - also mention that often it is not needed as it gets in the way of optimi
make dynamicly sized array thingy but using a class

new and delete
new[] and delete[]
reference counting to know when to clean up
defered copy until write for an object
    http://doc.qt.io/archives/qt-4.8/implicit-sharing.html#implicit-data-sharing
    lets you pass objects by value but keep the bulk of data frrom needing to be co
*/

//simple pointers
// showing a pointer is just a reference to a point in memory
```

```
39 void simple_pointer(){
40     int a;
41     a = 10;
42     printf("%d\n",a);
43 
44     int *p;
45     p = &a;
46     printf("%d\n",*p);
47 
48     *p = 25;
49     printf("%d\n",a);
50 }
51 
52 //=============================================================================
53 void pointers_walking_arrays(){
54     int b[4];
55     b[0]=2; b[1]=5; b[2]=8; b[3]=11;
56 
57     int *p = b;
58     printf("%d %d %d\n",*p,(*p)+1,*(p+1));
59 
60     for(int x=0; x<4; x++)
61         *(p+x) = 4*x;
62     printf("%d %d %d %d\n",b[0],b[1],b[2],b[3]);
63 }
64 
65 void pointers_walking_arrays2(){
66     //pointers can themselves be changed
67     int b[4];
68     b[0]=2; b[1]=4; b[2]=6; b[3]=8;
69     int *p = b;
70     p += 2;
71     printf("%d\n",*p);
72 
73     //pointers can be used to walk arrays of characters
74     char s[50]="Some Length";
75     char *sp = s;
76 
77     int counter=0;
78     while(*sp != 0){
79         counter+=1;
80         sp += 1;
```

```
81      }
82      printf("%d\n",counter);
83  }
84
85  //=============================================================================
86  int* __scope_of_variables(){
87      int x=25;
88      return &x;
89  }
90  void scope_of_variables(){
91      int *p=__scope_of_variables();
92      printf("%d\n",*p);
93  }
94
95  void scope_of_variables2(){
96      int *p;
97      { // some other scope
98          int x = 100;
99          p = &x;
100     }
101     printf("%d\n",*p);
102 }
103
104 //=============================================================================
105 void malloc_simple(){
106     int *p = (int*)malloc(4*sizeof(int));
107     for(int x=0; x<4; x++)
108         p[x] = x*4;
109     printf("%d %d %d %d\n",p[0],p[1],p[2],p[3]);
110     free(p);
111 }
112
113 int* __malloc_single(){
114     int *p = (int*)malloc(sizeof(int));
115     *p = 25;
116     return p;
117 }
118 void malloc_single(){
119     int *p = __malloc_single();
120     printf("%d\n",*p);
121     free(p);
122 }
```

```
123
124 void calloc_call(){
125     int *p = (int*)calloc(4,sizeof(int));
126     for(int x=0; x<4; x++)
127         p[x] = x*4;
128     printf("%d %d %d %d\n",p[0],p[1],p[2],p[3]);
129     free(p);
130 }
131
132 void realloc_example(){
133     int *p = (int*)calloc(2,sizeof(int));
134     p[0]=1; p[1]=1;
135
136     p = (int*)realloc(p,20*sizeof(int));
137     for(int x=2; x<20; x++)
138         p[x] = p[x-1] + p[x-2];
139     printf("%d\n",p[19]);
140     free(p);
141 }
142
143 void page_size(){
144     printf("%d\n",getpagesize());
145 }
146
147 //=============================================================================
148 class Simple{
149 public:
150     Simple(){printf("Constructed\n");}
151     ~Simple(){printf("Destructed\n");}
152 };
153
154 void object_malloc(){
155     Simple *p = (Simple*)malloc(sizeof(Simple));
156     free(p);
157 }
158 void object_new(){
159     Simple *p = new Simple;
160     delete p;
161 }
162
163 class HelloWorld{
164 public:
```

```
165     void print(){printf("Hello World\n");}
166 };
167 HelloWorld* __some_other_scope(){
168     return new HelloWorld;
169 }
170 void some_other_scope(){
171     HelloWorld *ppp = __some_other_scope();
172     ppp->print();
173     delete ppp;
174 }
175 
176 class Counter{
177     static int counter;
178     int our_value;
179 public:
180     Counter(){
181         our_value = counter;
182         counter++;
183     }
184     void print(){printf(" %d ",our_value);}
185 };
186 int Counter::counter = 0;
187 void object_array(){
188     Counter *array = new Counter[5];
189     for(int x=0; x<5; x++)
190         array[x].print();
191     printf("\n");
192     delete[] array;
193 }
194 
195 int main(){
196     // simple_pointer();
197 
198     // pointers_walking_arrays();
199     // pointers_walking_arrays2();
200 
201     // scope_of_variables();
202     // scope_of_variables2();
203 
204     // malloc_simple();
205     // malloc_single();
206     // calloc_call();
```

```
207     // realloc_example();
208     // page_size();
209
210     // object_malloc();
211     // object_new();
212     // some_other_scope();
213     object_array();
214     return 0;
215 }
```

B Code For Simple Dynamic Array

```c
#include <stdio.h>
#include <stdlib.h>

struct DynamicArray{
    void *data;
    int capacity;
    int length; // here for my own convenience
};

struct DynamicArray* make_array(int size){
    if(size<=0)
        size = 1;
    struct DynamicArray *info = (struct DynamicArray*)malloc(sizeof(struct DynamicArr
    info->data = malloc(size);
    info->capacity = size;
    info->length = 0;
}
void delete_array(struct DynamicArray* info){
    free(info->data);
    free(info);
}
void resize_array(struct DynamicArray* info,int size){
    if(size==0) size = 1;
    if(size<0){
        int delta = info->capacity;
        if(delta > 128) delta = 128;
        info->data = realloc(info->data,delta+info->capacity);
        info->capacity += delta;
    }else{
        info->data = realloc(info->data,size);
        info->capacity = size;
    }
}

int main(){
    //Make array and fill with 50s
    struct DynamicArray *array = make_array(10*sizeof(int));
    for(int x=0; x<10; x++)
        ((int*)array->data)[x] = 50;
```

```
40      array->length = 10; // just to help me bookkeep
41
42      //lets just check that things are working as we expect
43      //print all 10 indicies
44      for(int x=0; x<array->length; x++){
45          printf(" %3d ",((int*)array->data)[x]);
46      }
47      printf("\n");
48
49      //now lets make the array larger
50      resize_array(array,30*sizeof(int));
51      for(int x=0; x<10; x++)
52          ((int*)array->data)[x+10] = 100;
53      for(int x=0; x<10; x++)
54          ((int*)array->data)[x+20] = 150;
55      array->length = 30; // just to help me bookkeep
56
57      //lets just check that things are working as we expect
58      for(int x=0; x<array->length; x++){
59          if(x%10 == 0)
60              printf("\n");
61          printf(" %3d ",((int*)array->data)[x]);
62      }
63      printf("\n");
64
65      //now lets make the array smaller
66      resize_array(array,20*sizeof(int));
67      array->length = 20; // just to help me bookkeep
68
69      //lets just check that things are working as we expect
70      for(int x=0; x<array->length; x++){
71          if(x%10 == 0)
72              printf("\n");
73          printf(" %3d ",((int*)array->data)[x]);
74      }
75      printf("\n");
76
77      return 0;
78  }
```

C Code For Object Based Dynamic Array

```
#include <stdio.h>
#include <stdlib.h>

template <typename T>
class vector{
    T* data;
    int length,capacity;
public:
    vector(){
        data = (T*)malloc(sizeof(T));
        length = 0;
        capacity = 1;
    };
    ~vector(){delete[] data;}
    void push_back(T& item);
    void push_back(T item);
    void pop_back();
    T& operator[](int index);
    int size(){return length;}
};
template<typename T>
void vector<T>::push_back(T& item){
    if(length >= capacity){
        //need more room, time to realloc()
        int delta = capacity;
        delta %= 128; //only get bigger by steps of 128 at most
        data = (T*)realloc(data,(delta+capacity) * sizeof(T));
        capacity += delta;
    }
    data[length] = item;
    length++;
}
template<typename T>
void vector<T>::push_back(T item){
    if(length >= capacity){
        //need more room, time to realloc()
        int delta = capacity;
        delta %= 128; //only get bigger by steps of 128 at most
        data = (T*)realloc(data,(delta+capacity) * sizeof(T));
```

```
40          capacity += delta;
41      }
42      data[length] = item;
43      length++;
44  }
45
46  template<typename T>
47  void vector<T>::pop_back(){
48      length--;
49  }
50
51  template<typename T>
52  T& vector<T>::operator[](int index){
53      return data[index];
54  }
55
56  class Counter{
57      static int counter;
58  public:
59      int our_value;
60      Counter(){
61          our_value = counter;
62          counter++;
63      }
64      void print(){printf(" C%d ",our_value);}
65  };
66  int Counter::counter = 0;
67
68  int main(){
69      // vector<int> test;
70      // //add items
71      // for(int x=0; x<10; x++)
72      //     test.push_back(x);
73      // //read items
74      // for(int x=0; x<test.size(); x++)
75      //     printf(" %d ",test[x]);
76      // printf("\n");
77
78      // //remove items
79      // for(int x=0; x<5; x++)
80      //     test.pop_back();
81      // //read items
```

```
82      // for(int x=0; x<test.size(); x++)
83      //     printf(" %d ",test[x]);
84      // printf("\n");
85
86      // //change items
87      // for(int x=0; x<5; x++)
88      //     test[x] = (x+1) * 5;
89      // //read items
90      // for(int x=0; x<test.size(); x++)
91      //     printf(" %d ",test[x]);
92      // printf("\n");
93
94      vector<Counter> test;
95      //add items
96      for(int x=0; x<10; x++)
97          test.push_back(Counter());
98      //read items
99      for(int x=0; x<test.size(); x++)
100         test[x].print();
101     printf("\n");
102
103     //remove items
104     for(int x=0; x<5; x++)
105         test.pop_back();
106     //read items
107     for(int x=0; x<test.size(); x++)
108         test[x].print();
109     printf("\n");
110
111     //change items
112     for(int x=0; x<5; x++)
113         test[x].our_value = (x+1) * 5;
114     //read items
115     for(int x=0; x<test.size(); x++)
116         test[x].print();
117     printf("\n");
118
119     return 0;
120 }
```